Probabilistic Programming Languages

Guillaume Baudart

MPRI 2024-2025

Reminder: Exact Inference

Probabilistic Programming Languages

Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

- Prior: $z \sim \mathit{Uniform}(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim \mathit{Bernoulli}(z)$
- Posterior: $p(z \mid x_1, \ldots, x_n)$?

Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

Probabilistic model

- Prior: $z \sim \textit{Uniform}(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim \textit{Bernoulli}(z)$
- Posterior: $p(z \mid x_1, \ldots, x_n)$?

$$
\begin{aligned}
p(x_1, \ldots, x_n \mid z) &= \prod_{i=1}^{n} p(x_i \mid z) \\
&= \prod_{i=1}^{n} z^{x_i} \ (1-z)^{1-x_i} \\
&= z^{\sum_{i=1}^{n} x_1} \ (1-z)^{\sum_{i=1}^{n} (1-x_i)} \\
&= z^{\#\text{heads}} \ (1-z)^{\#\text{tails}}
\end{aligned}
$$

$$
\begin{aligned}
p(z \mid x_1, \ldots, x_n) &= \frac{p(x_1, \ldots, x_n \mid z) p(z)}{p(x_1, \ldots, x_n)} \\
&= \frac{p(x_1, \ldots, x_n \mid z) p(z)}{\int_z p(x_1, \ldots, x_n \mid z)}
\end{aligned}
$$

$$
\begin{aligned}
p(z \mid x_1, \ldots, x_n) &= \frac{z^{\#\text{heads}} \ (1-z)^{\#\text{tails}}}{\int_z z^{\#\text{heads}} \ (1-z)^{\#\text{tails}}} \\
&= \frac{z^{\#\text{heads}} \ (1-z)^{\#\text{tails}}}{B(\#\text{heads} + 1, \ \#\text{tails} + 1)} \\
&= \text{pdf}(\textit{Beta}(\#\text{heads} + 1, \ \#\text{tails} + 1))
\end{aligned}
$$

Example: Coin

Consider a series of coin tosses

- Observations: head or tail
- Each toss is independant
- What is the probability of getting head at the next toss?

$$
\begin{aligned}
p(x_1, \ldots, x_n \mid z) &= \prod_{i=1}^{n} p(x_i \mid z) \\
&= \prod_{i=1}^{n} z^{x_i} \ (1-z)^{1-x_i} \\
&= z^{\sum_{i=1}^{n} x_1} \ (1-z)^{\sum_{i=1}^{n}(1-x_i)} \\
&= z^{\#\text{heads}} \ (1-z)^{\#\text{tails}}
\end{aligned}
$$

Probabilistic model

- Prior: $z \sim \mathit{Uniform}(0, 1)$
- Observations: for $i \in [1, n]$, $x_i \sim \mathit{Bernoulli}(z)$
- Posterior: $p(z \mid x_1, \ldots, x_n)$?

$$
\begin{aligned}
p(z \mid x_1, \ldots, x_n) &= \\
&= \\
&= \mathit{eta}(\#\text{heads} + 1, \ \#\text{tails} + 1))
\end{aligned}
$$

What if the model is much more complex?

What if we use arbitrary control flow?

Can we compute the posterior automatically?

Semi-Symbolic Inference

Probabilistic Programming Languages

Semi-symbolic inference

Approximate inference can be impractical

- Require lot of computing power
- Can yield poor approximation

Exact inference is often possible

Rao-Blackwellized inference

- Each execution of the model maintains symbolic distributions
- Perform as much exact computation as possible
- Fall back to sampling when symbolic computation fails

Key elements

- Marginalization: remove dependencies
- Conditioning: incorporate observations

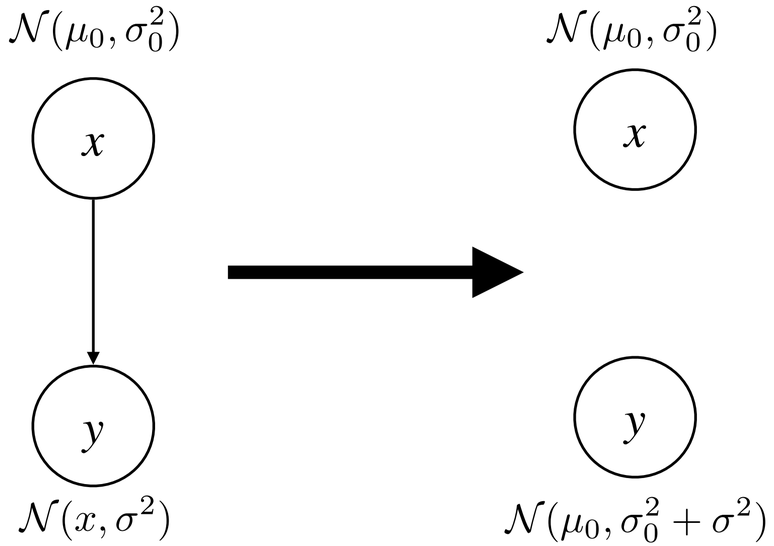
Marginalization

Remove dependencies

- Parent distribution: $p(x)$
- Child distribution: $p(x \mid y)$
- Marginal: $p(x) = \int p(x \mid y)p(x)$

Gaussians

- Parent distribution: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Child distribution: $(x \mid y) \sim \mathcal{N}(x, \sigma^2)$
- Marginal: $y \sim \mathcal{N}(\mu_0, \sigma_0^2 + \sigma^2)$



Conjugate priors

Bayesian Inference: learn parameters from data

- Latent parameter θ
- Observed data $x_1, \ldots, x_n$

$$p(\theta \mid x_1, \ldots x_n) = \frac{p(\theta)\ p(x_1, \ldots, x_n \mid \theta)}{p(x_1, \ldots, x_n)} \qquad \text{(Bayes' theorem)}$$

$$\propto p(\theta)\ p(x_1, \ldots, x_n \mid \theta) \qquad \text{(Data are constants)}$$

If the posterior $p(\theta \mid x)$ is in the same distribution family as the prior $p(\theta)$

- Prior and posterior are called *conjugate distributions* w.r.t., the likelihood
- The prior is the *conjugate prior* for the likelihood function

Conjugate priors examples

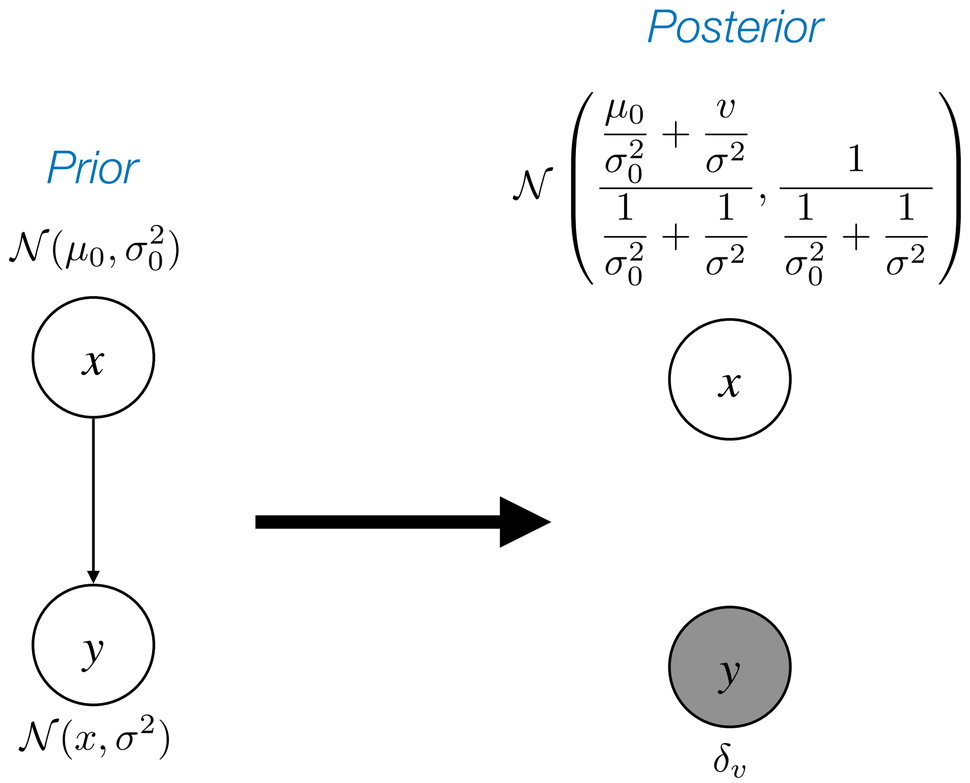
Beta-Bernoulli

- Prior: $x \sim \mathit{Beta}(a, b)$
- Likelihood: $(y \mid x) \sim \mathit{Bernoulli}(x)$
- Posterior: $(x \mid x = v) \sim \mathit{Beta}(a + v, b + (1 - v))$

Gaussians

- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\begin{aligned} \mu_1 &= \left(\tfrac{1}{\sigma_0^2} + \tfrac{1}{\sigma^2}\right)^{-1} \left(\tfrac{\mu_0}{\sigma_0^2} + \tfrac{v}{\sigma^2}\right) \\ \sigma_1^2 &= \left(\tfrac{1}{\sigma_0^2} + \tfrac{1}{\sigma^2}\right)^{-1} \end{aligned}$$



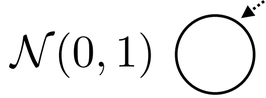
Try it in BYO-PPL!

Delayed Sampling

Probabilistic Programming Languages

Delayed sampling

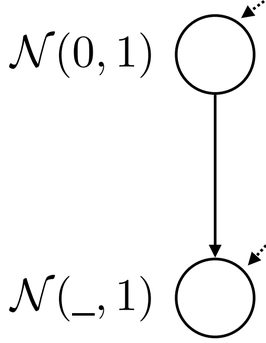
```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  a
```



Murray et al. 2018

Delayed sampling

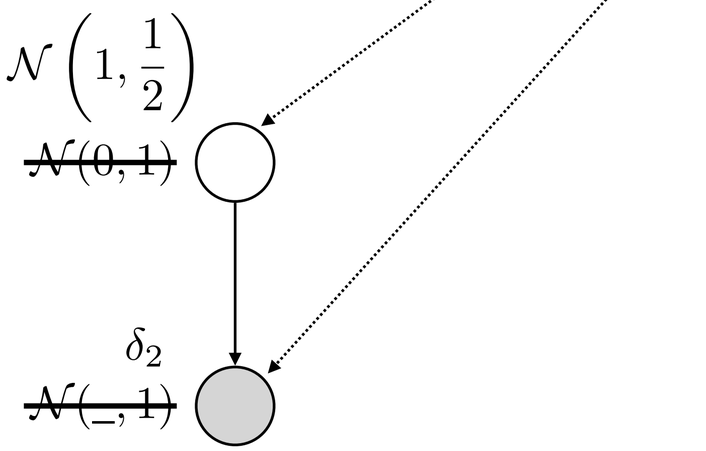
```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  a
```



Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  a
```



Conditionning

Gaussians

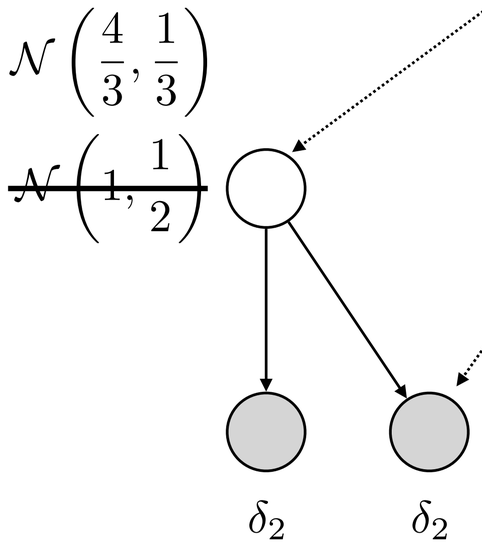
- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2}\right)$$
$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1}$$

Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  a
```



Conditionning

Gaussians

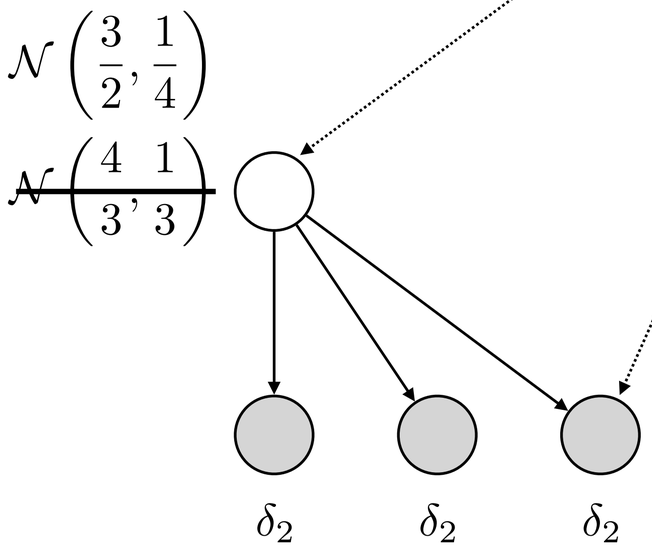
- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2}\right)$$
$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1}$$

Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  observe (gaussian z 1.) 2.;
  a
```



Conditionning

Gaussians

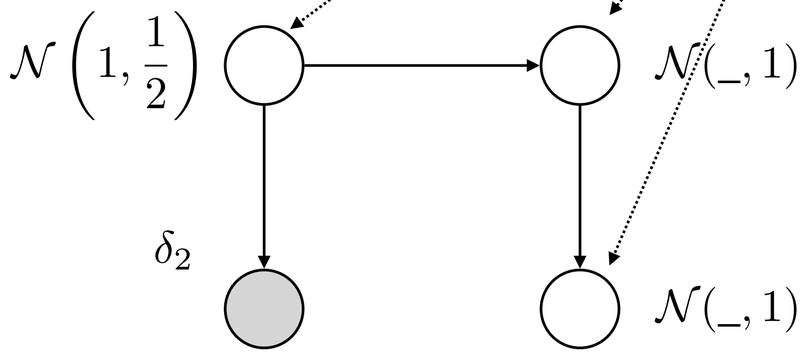
- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\mu_1 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2}\right)$$
$$\sigma_1^2 = \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1}$$

Murray et al. 2018

Delayed sampling

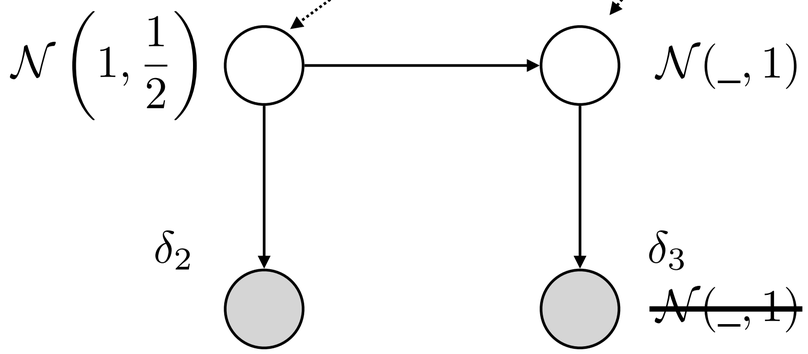
```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian a 1.) 2.;
  let b = sample gaussian a 1. in
  observe (gaussian b 1.) 2.;
  b
```



Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian a 1.) 2.;
  let b = sample gaussian a 1. in
  observe (gaussian b 1.) 3.;
  b
```

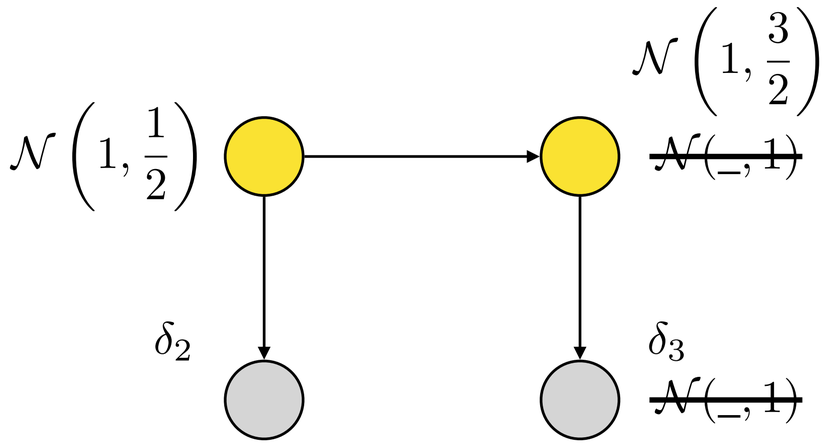


Conditional distribution
Cannot condition yet.

Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian a 1.) 2.;
  let b = sample gaussian a 1. in
  observe (gaussian b 1.) 3.
```



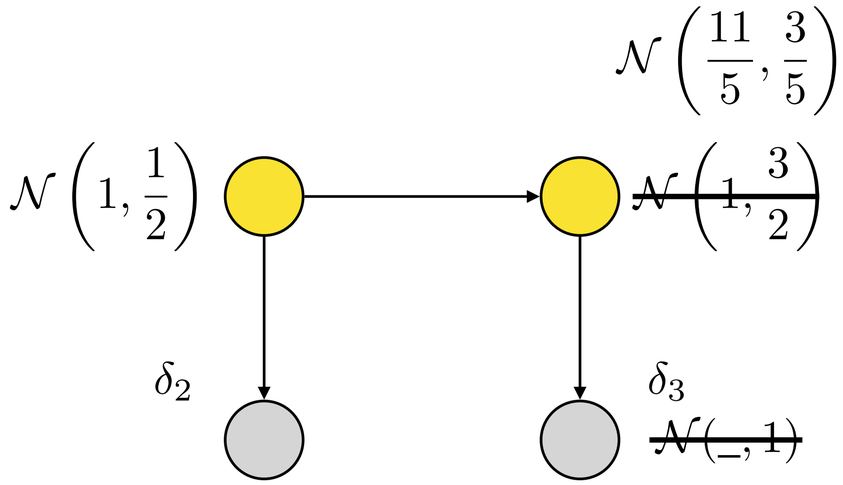
Marginalization

Gaussians
- Parent distribution: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Child distribution: $(x \mid y) \sim \mathcal{N}(x, \sigma^2)$
- Marginal: $y \sim \mathcal{N}(\mu_0, \sigma_0^2 + \sigma^2)$

Murray et al. 2018

Delayed sampling

```
let gauss () =
  let a = sample gaussian 0. 1. in
  observe (gaussian a 1.) 2.;
  let b = sample gaussian a 1. in
  observe (gaussian b 1.) 3.
```



Conditionning

Gaussians

- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\begin{aligned} \mu_1 &= \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2}\right) \\ \sigma_1^2 &= \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \end{aligned}$$

Murray et al. 2018

Delayed sampling: summary

`infer model data`

- Launch n independent executions (approximate inference)
- Maintain a graph (Bayesian model for the current execution)
- Returns a mixture of symbolic distributions

`sample d`

- add a node to the graph (random variable)
- keep track of dependencies

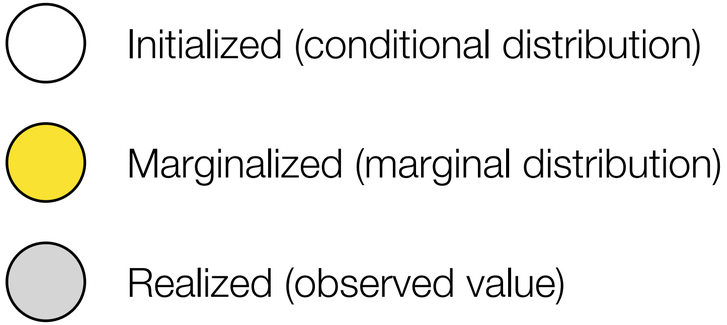
`observe`

- add a node to the graph (observed random variable)
- marginalize parent (recursively until the root)
- condition the parent on the observation

If error (conditioning or marginalization fails)

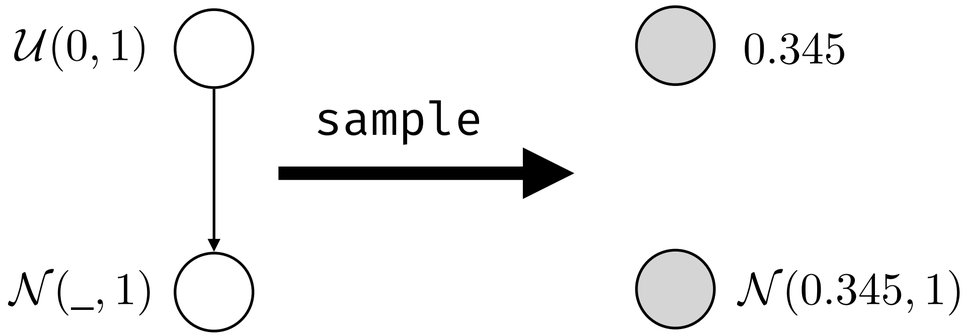
- Sample parent
- Update the children

Node state a runtime

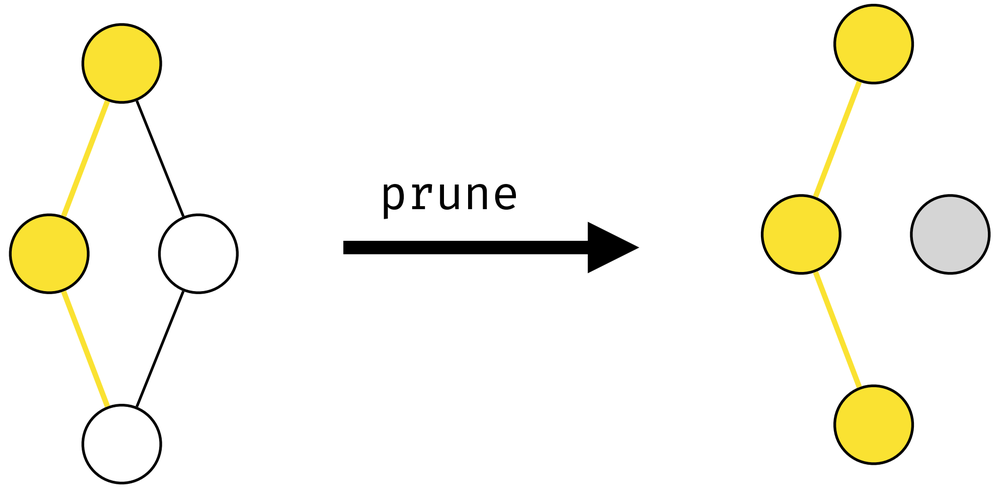


Murray et al. 2018

Delayed sampling: actual sampling



no conjugacy



single m-path

Delayed sampling is limited to forests.
Can we do better?

Semi-symbolic inference

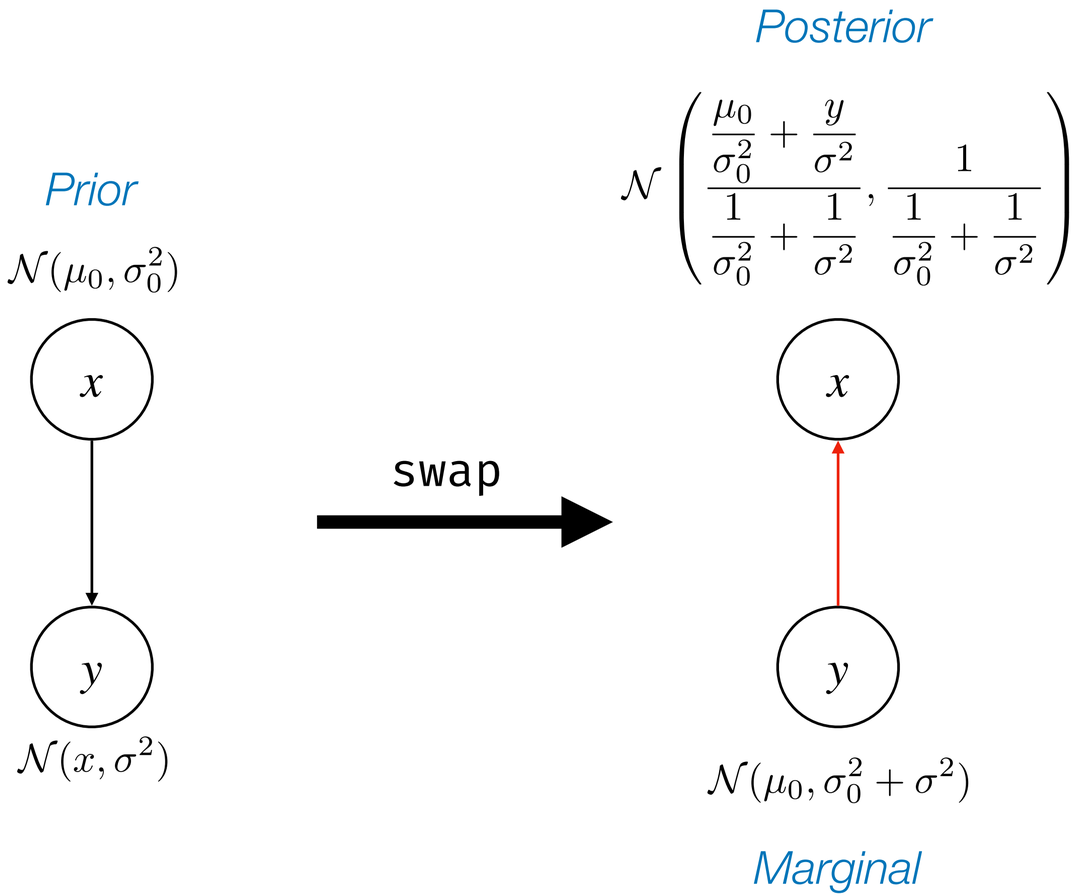
Probabilistic Programming Languages

Semi-symbolic inference

Gaussians

- Prior: $x \sim \mathcal{N}(\mu_0, \sigma_0^2)$
- Likelihood: $(y \mid x) \sim \mathcal{N}(x, \sigma^2)$
- Posterior: $(y \mid x = v) \sim \mathcal{N}(\mu_1, \sigma_1^2)$

$$\begin{aligned} \mu_1 &= \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \left(\frac{\mu_0}{\sigma_0^2} + \frac{v}{\sigma^2}\right) \\ \sigma_1^2 &= \left(\frac{1}{\sigma_0^2} + \frac{1}{\sigma^2}\right)^{-1} \end{aligned}$$



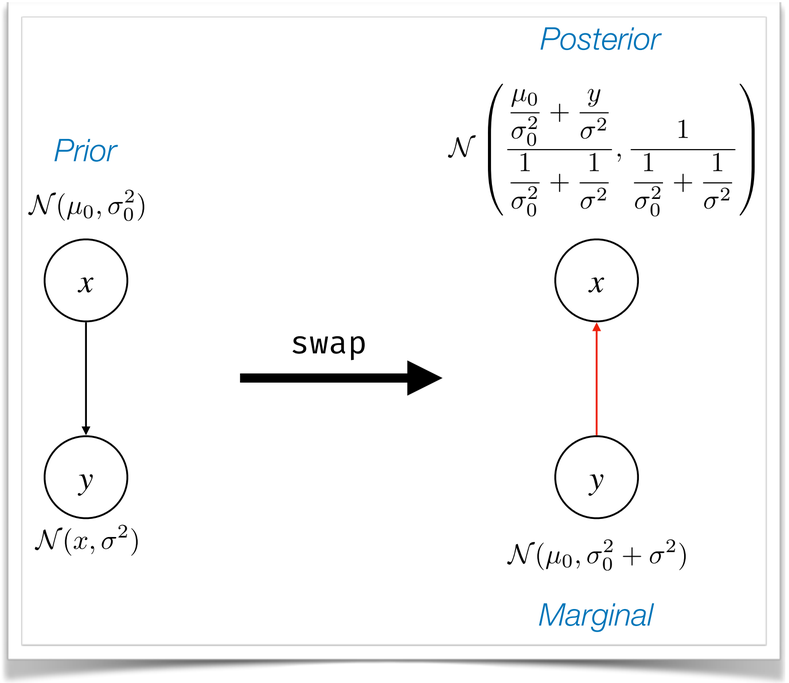
Semi-symbolic inference

Main idea

- Maintain symbolic expressions involving random variables
- Each operation triggers a sequence of swaps
- Swap change the dependency between two variables but preserves the symbolic state
- Only root nodes (with no parent) can be observed

Closed families

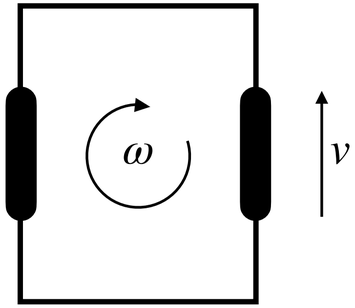
- A swap always succeed and the resulting state remains in the family.
- E.g., linear Gaussians models, and finite discrete models.

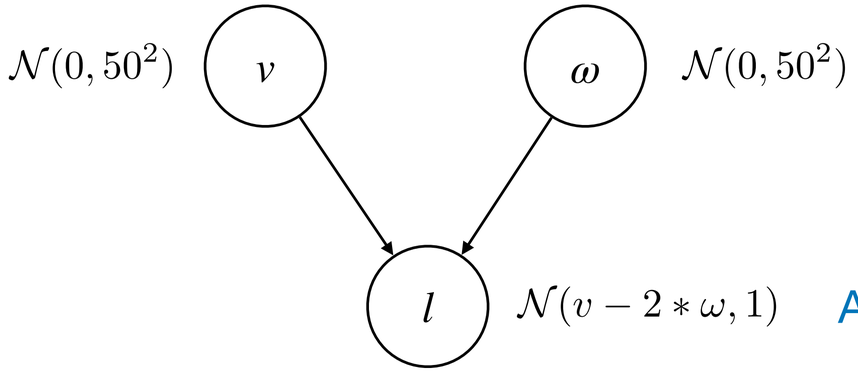


Atkinson et al. 2022

Semi-symbolic inference

```
let wheels(l_rate, r_rate) =
  let omega = sample (gaussian 0. 50.0) in
  let vel = sample (gaussian 0. 50.0) in
  observe (gaussian(vel - 2.0 * omega, 1.0) l_rate;
  observe (gaussian(vel + 2.0 * omega, 1.0) r_rate;
  omega, vel
```



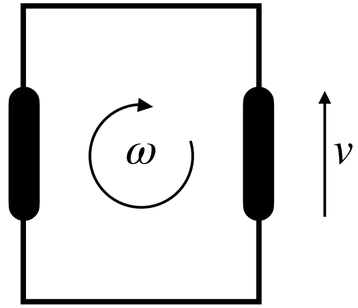


Affine Gaussians

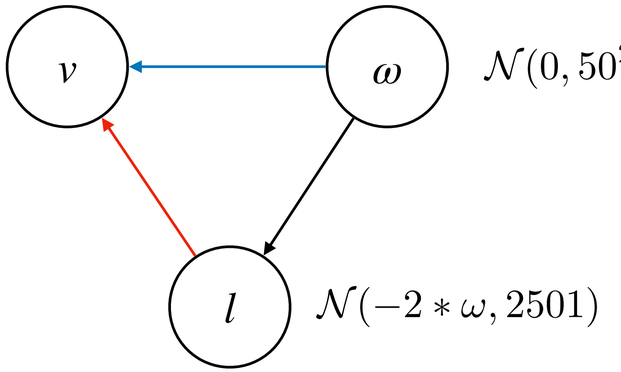
Atkinson et al. 2022

Semi-symbolic inference

```
let wheels(l_rate, r_rate) =
  let omega = sample (gaussian 0. 50.0) in
  let vel = sample (gaussian 0. 50.0) in
  observe (gaussian(vel - 2.0 * omega, 1.0) l_rate;
  observe (gaussian(vel + 2.0 * omega, 1.0) r_rate;
  omega, vel
```



$$\mathcal{N}\left(\frac{2500}{2501} * l + \frac{5000}{2501} * w, \frac{2500}{2501}\right)$$

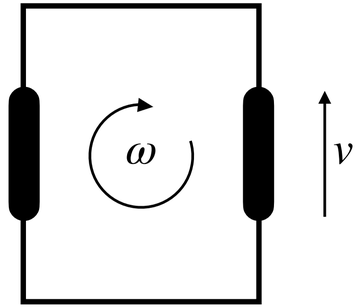


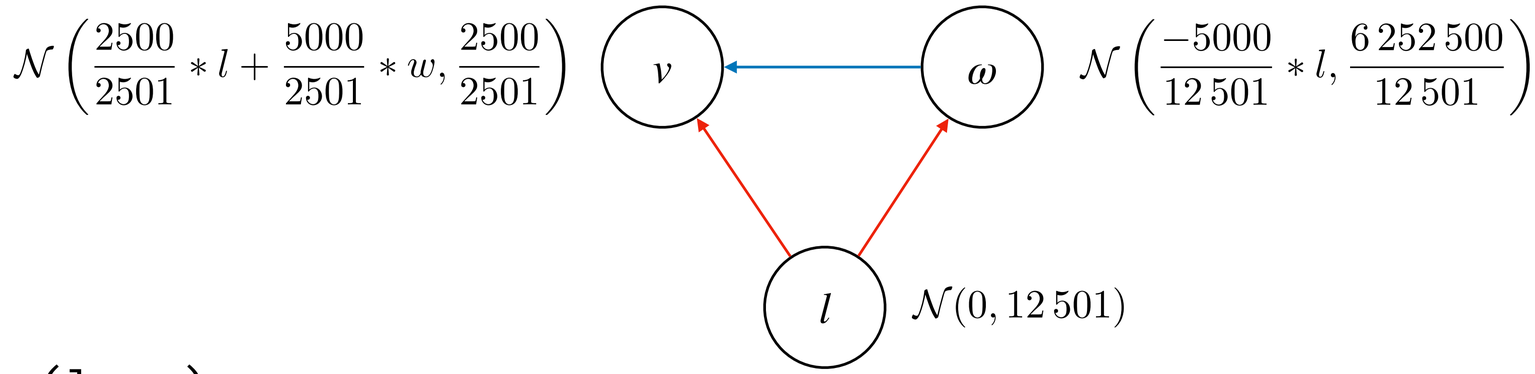

```
swap(l, v)
```


Atkinson et al. 2022

Semi-symbolic inference

```
let wheels(l_rate, r_rate) =
  let omega = sample (gaussian 0. 50.0) in
  let vel = sample (gaussian 0. 50.0) in
  observe (gaussian(vel - 2.0 * omega, 1.0) l_rate;
  observe (gaussian(vel + 2.0 * omega, 1.0) r_rate;
  omega, vel
```



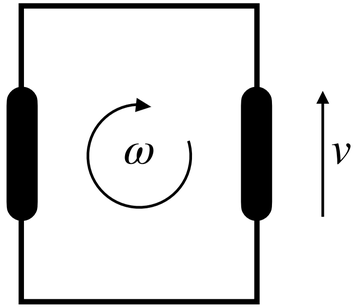



```
swap(l, w)
```

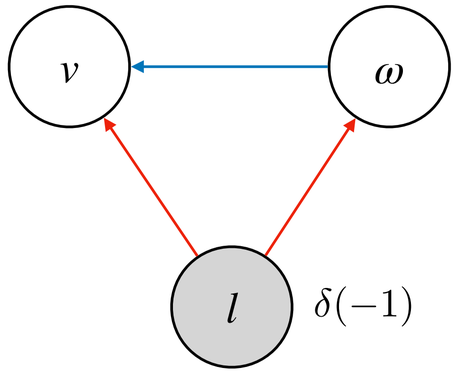

Atkinson et al. 2022

Semi-symbolic inference

```
let wheels(l_rate, r_rate) =
  let omega = sample (gaussian 0. 50.0) in
  let vel = sample (gaussian 0. 50.0) in
  observe (gaussian(vel - 2.0 * omega, 1.0) l_rate;
  observe (gaussian(vel + 2.0 * omega, 1.0) r_rate;
  omega, vel
```



$$\mathcal{N}\left(\frac{-2500}{2501} + \frac{5000}{2501} * w, \frac{2500}{2501}\right)$$



$$\mathcal{N}\left(\frac{5000}{12\,501}, \frac{6\,252\,500}{12\,501}\right)$$

```
value(l)
```


Atkinson et al. 2022

Semi-symbolic: summary

`infer model data`

- Launch n independent executions (approximate inference)
- Maintain a graph (Bayesian model for the current execution)
- Returns a mixture of symbolic distributions

`sample d`

- add a node to the graph (random variable)
- keep track of dependencies

`observe`

- add a node to the graph (observed random variable)
- swaps to turn the node into a root
- condition an simplify the graph

If error (swap fails)

- Sample (some) parents
- Update the children

More general than delayed sampling

Atkinson et al. 2022

Reactive Probabilistic Programming

Probabilistic Programming Languages

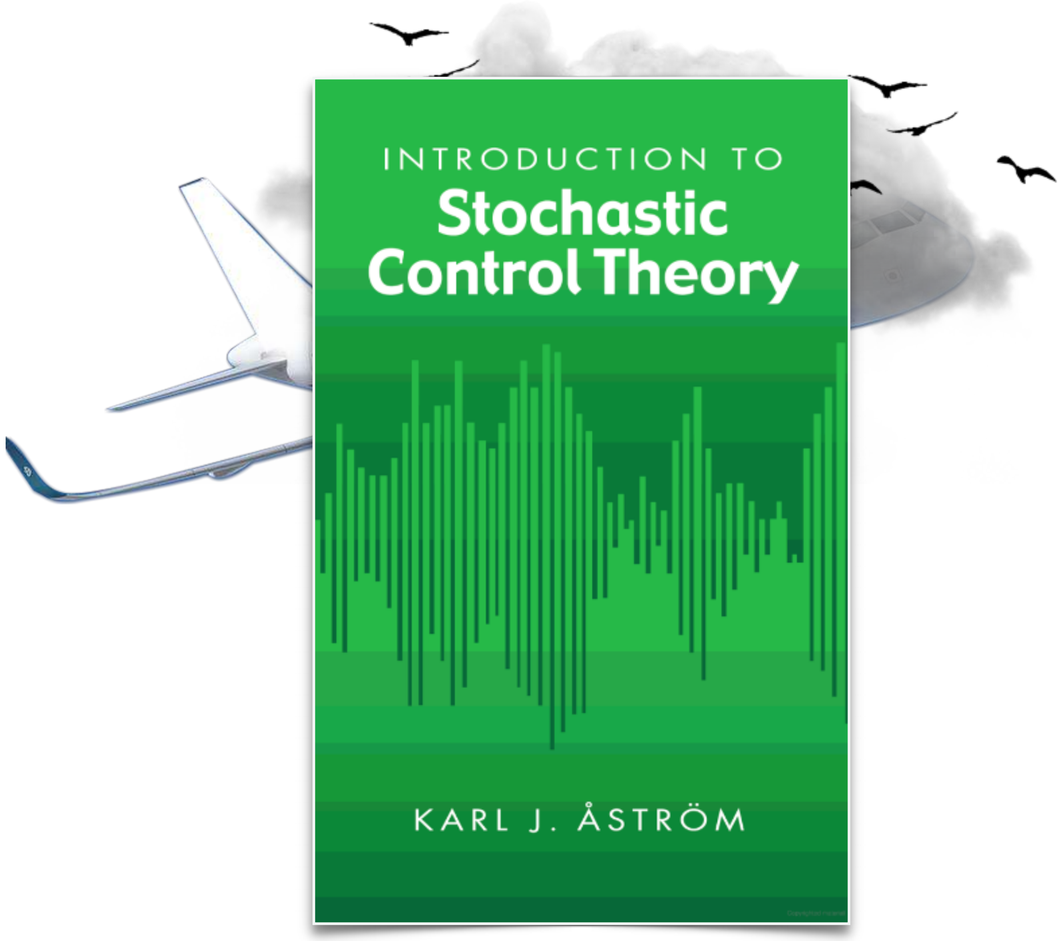
Uncertainty in embedded systems

Synchronous languages
- High-level specification language
- Generate correct-by-construction embedded code
- Industrial tool: ANSYS Scade

Challenges
- Noisy environment, perceived through noisy sensors
- Interaction with other autonomous entities

Existing approaches
- Manually implement stochastic controller: *Can be error prone*
- Offline statistical tests: *Requires up-to-date offline data*

Reactive Probabilistic Programming
- Synchronous languages with probabilistic constructs
- Make the probabilistic model explicit
- Automatically learn posterior distributions from observations

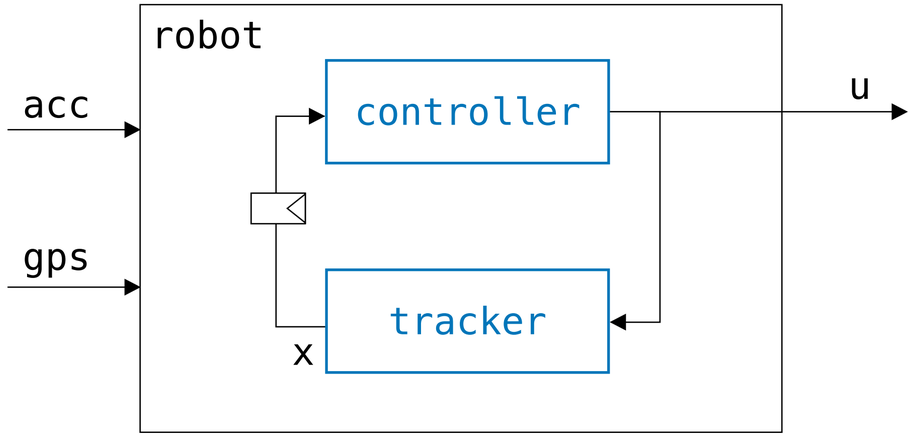


Reactive systems

Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

State: `x:` $(position \times velocity \times acceleration)$



Reactive probabilistic systems

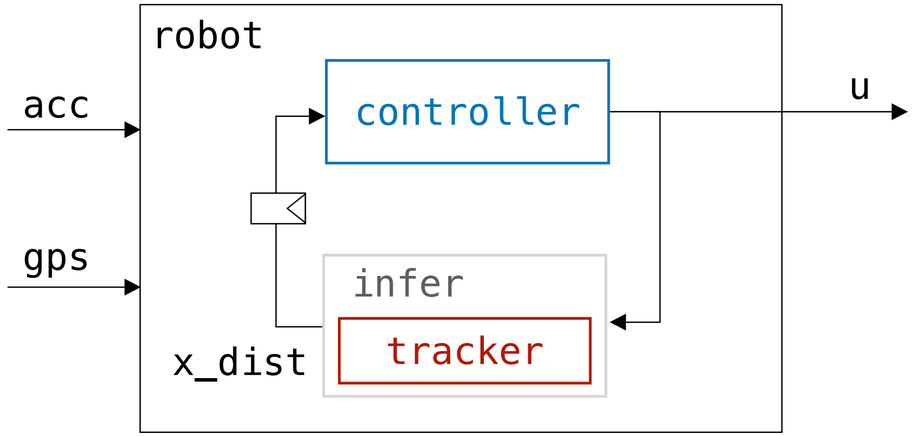
Synchronous data-flow languages and block diagrams

- Signal: stream of values
- System: stream processor

ProbZelus: add support to deal with uncertainty

- Extend a synchronous language
- Parallel composition: deterministic/probabilistic
- Inference-in-the-loop
- Streaming inference

State: `x_dist:` $(position \times velocity \times acceleration)$ `dist`



Synchronous Programming

Reactive Probabilistic Programming

Example: tracker

latent

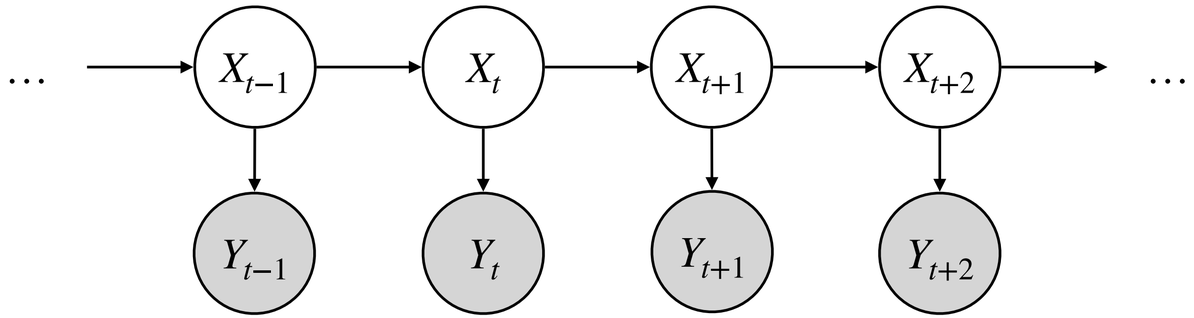
observed

Model

- Linear motion: $X_k \sim \mathcal{N}(FX_{k-1}, Q)$
- Observation: $Y_k \sim \mathcal{N}(HX_k, R)$

E.g., with Q and R constant noise matrices

- $X_k = \begin{pmatrix} p_k \\ v_k \end{pmatrix}$ (position, velocity)
- $F = \begin{pmatrix} 1 & dt \\ 0 & 1 \end{pmatrix}$ (discrete integration)
- $H = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ (projection)



$$
\begin{aligned}
X_k^{\text{pred}} &= FX_{k-1}^{\text{est}} \\
X_k^{\text{est}} &= X_k^{\text{pred}} + K_k(Y_k - HX_k^{\text{pred}}) \\
S_k &= (R + HP_k^{\text{pred}}H^T)^{-1} \\
K_k &= P_k^{\text{pred}}H^T S_k \\
P_k^{\text{pred}} &= Q + FP_{k-1}^{\text{est}}F^T \\
P_k^{\text{est}} &= P_k^{\text{pred}} - K_k H P_k^{\text{pred}}
\end{aligned}
$$

Solution: Kalman filter

Reactive synchronous programming

Dataflow synchronous programming

- Set of stream equations
- Discrete logical time steps
- At each step, compute the current value given inputs and previous values

```
let node kalman(y) = x_est where
  rec x_pred = f * (x0 → pre x_est)
  and x_est  = x_pred + k * (y - h * x_pred)
  and s      = r + h * p_pred * (transpose h)
  and k      = p_pred * (transpose h) * (inv s)
  and p_pred = q + f * (p0 → pre p_est) * (transpose f)
  and p_est  = p_pred - k * h * p_pred
```

$$
\begin{aligned}
X_k^{\text{pred}} &= F X_{k-1}^{\text{est}} \\
X_k^{\text{est}} &= X_k^{\text{pred}} + K_k(Y_k - H X_k^{\text{pred}}) \\
S_k &= R + H P_k^{\text{pred}} H^T \\
K_k &= P_k^{\text{pred}} H^T S_k^{-1} \\
P_k^{\text{pred}} &= Q + F P_{k-1}^{\text{est}} F^T \\
P_k^{\text{est}} &= P_k^{\text{pred}} - K_k H P_k^{\text{pred}}
\end{aligned}
$$

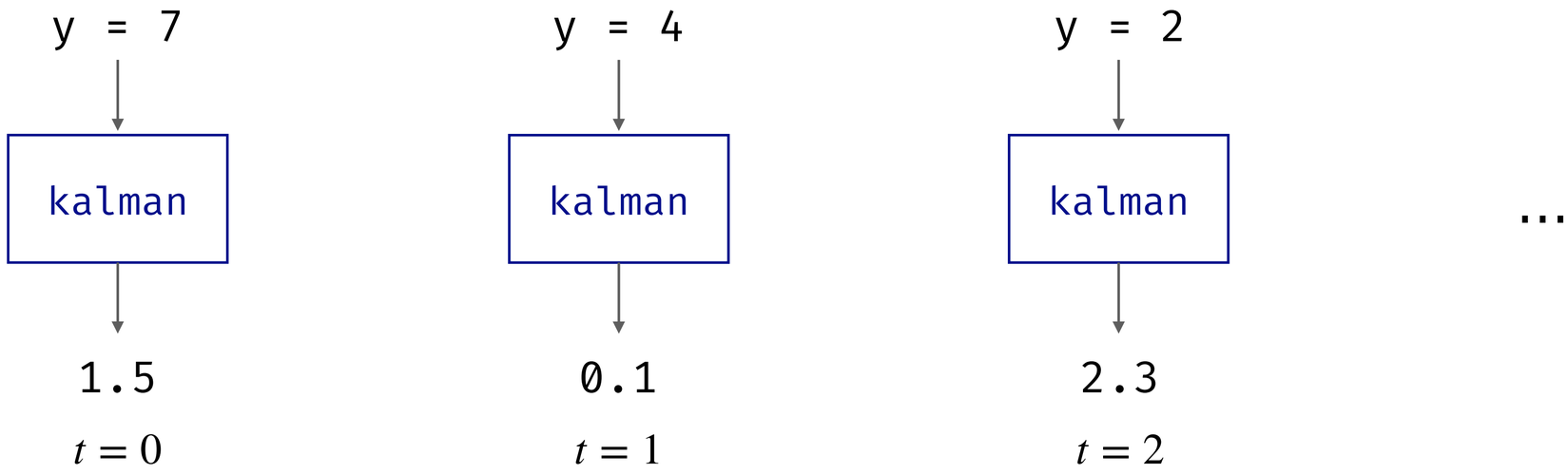
Solution: Kalman filter

Bourke, Pouzet 2013

Reactive synchronous programming

```
let node kalman(y) = x_est where
  rec x_pred = f * (x0 → pre x_est)
  and x_est  = x_pred + k * (y - h * x_pred)
  and s      = r + h * p_pred * (transpose h)
  and k      = p_pred * (transpose h) * (inv s)
  and p_pred = q + f * (p0 → pre p_est) * (transpose f)
  and p_est  = p_pred - k * h * p_pred
```

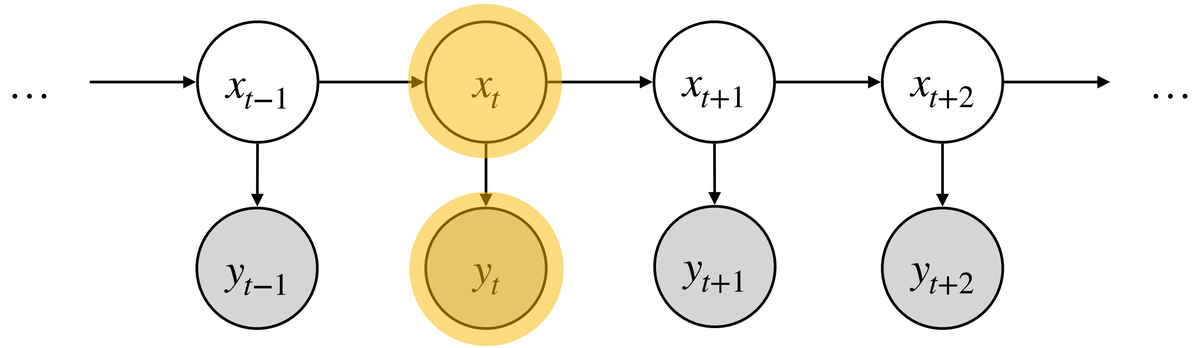
What if the assumptions change?
What if the model is not linear?

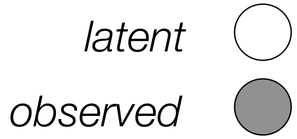


Reactive probabilistic programming

Probabilistic constructs

- `x = sample(d)`: introduce a random variable `x` of distribution `d`
- `observe(d, y)`: condition on the fact that `y` was sampled from `d`
- `infer m y`: compute posterior distribution of `m` given `y`



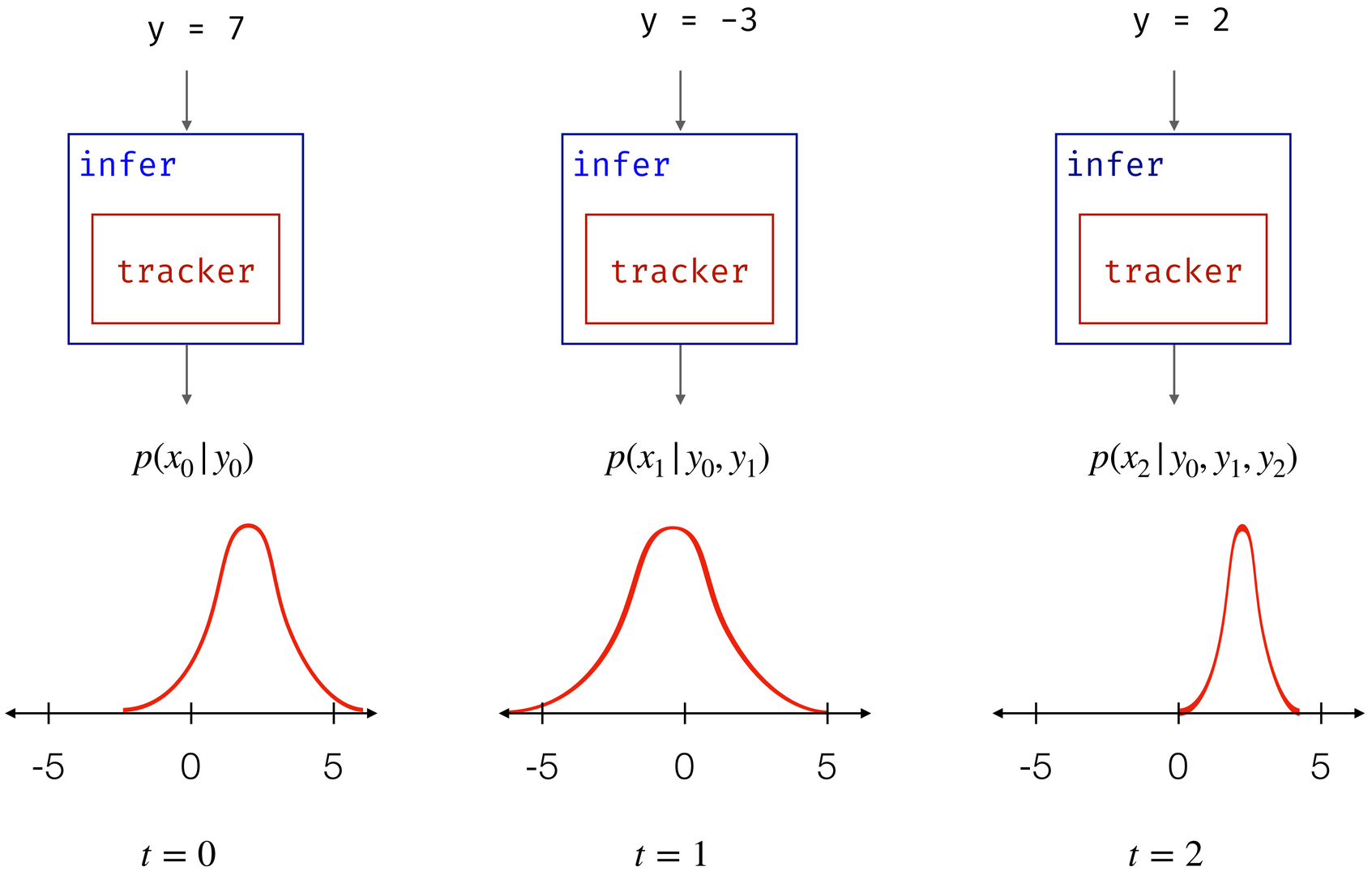



```
let proba tracker (y) = x where
  rec x = x0 → sample(mv_gaussian(f * (pre x), q))
  and () = observe(mv_gaussian(h * x, r), y)
```

Model

- Linear motion: $X_k \sim \mathcal{N}(FX_{k-1}, Q)$
- Observation: $Y_k \sim \mathcal{N}(HX_k, R)$

Reactive probabilistic programming



Demo

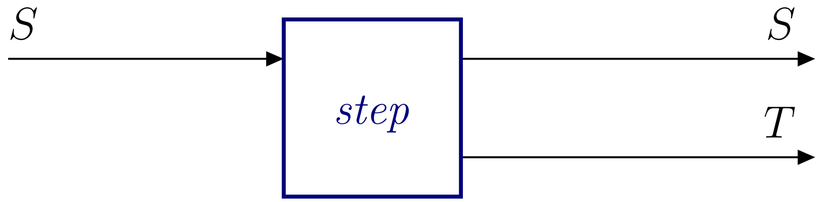
Reactive semantics

Reactive Probabilistic Programming

Deterministic streams

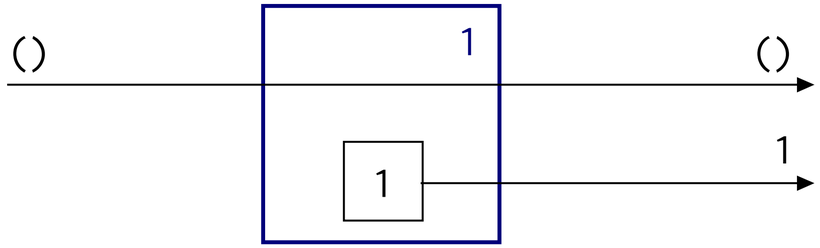
Initial state, transition function

$CoStream(T, S) = S \times (S \to S \times T)$



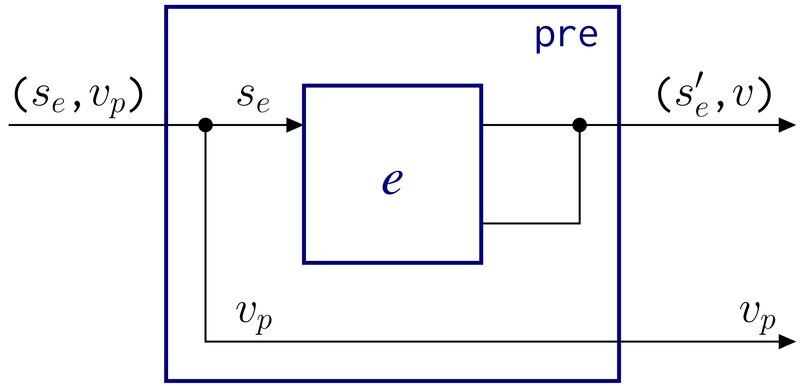
Constant 1: `unit` $\to$ `unit` $\times$ `int`

- Initial state: the value of type unit
- Step function: return the constant 1



Unit delay `pre` e: $(S \times T) \times (S \times T \to (S \times T) \times T)$

- Initial state: the initial state of e and default value
- Step function: the result of e is stored in the state and returned at the next iteration

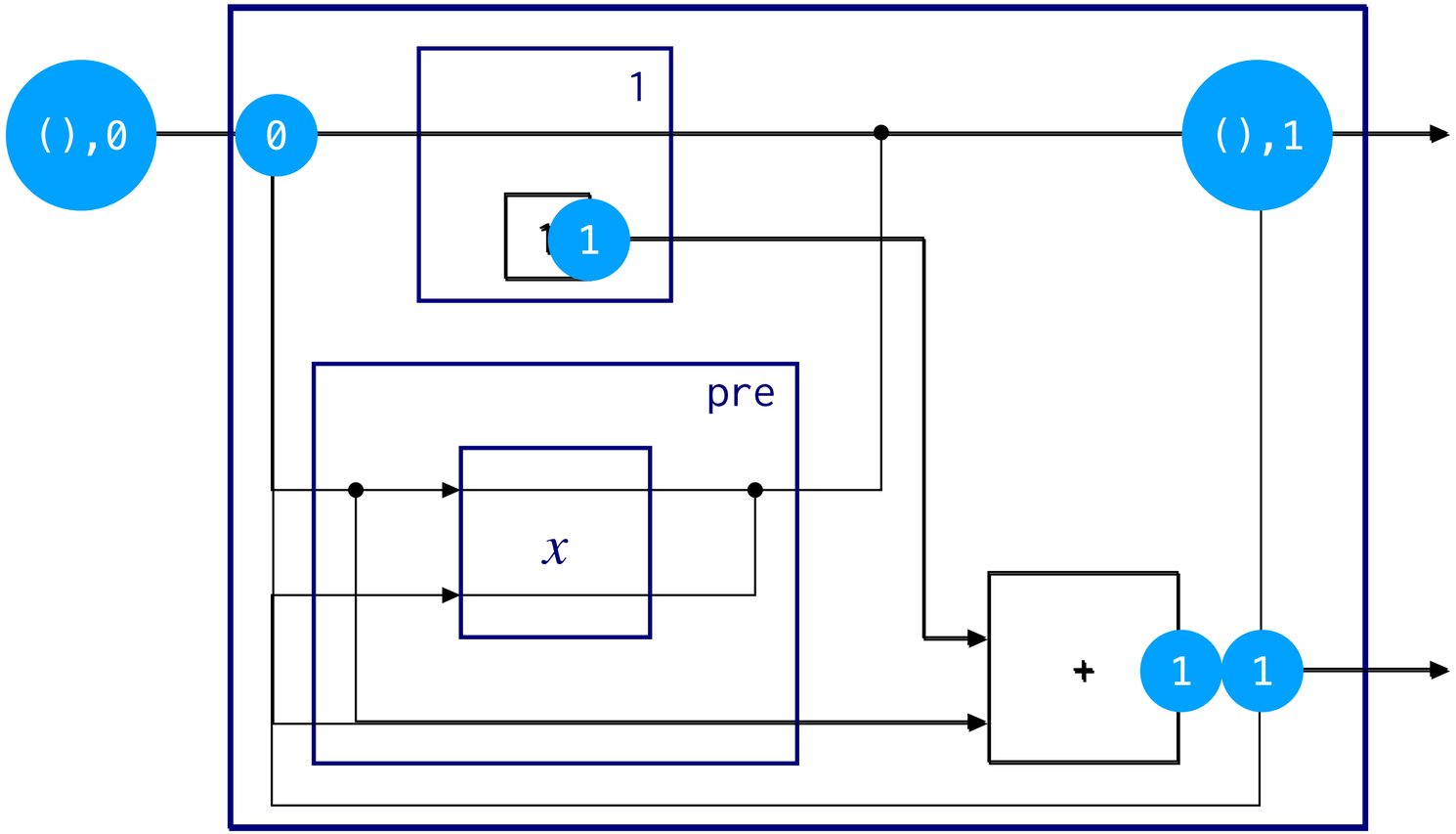


Caspi & Pouzet. 1998

Example

```
rec x = 1 + pre x
```

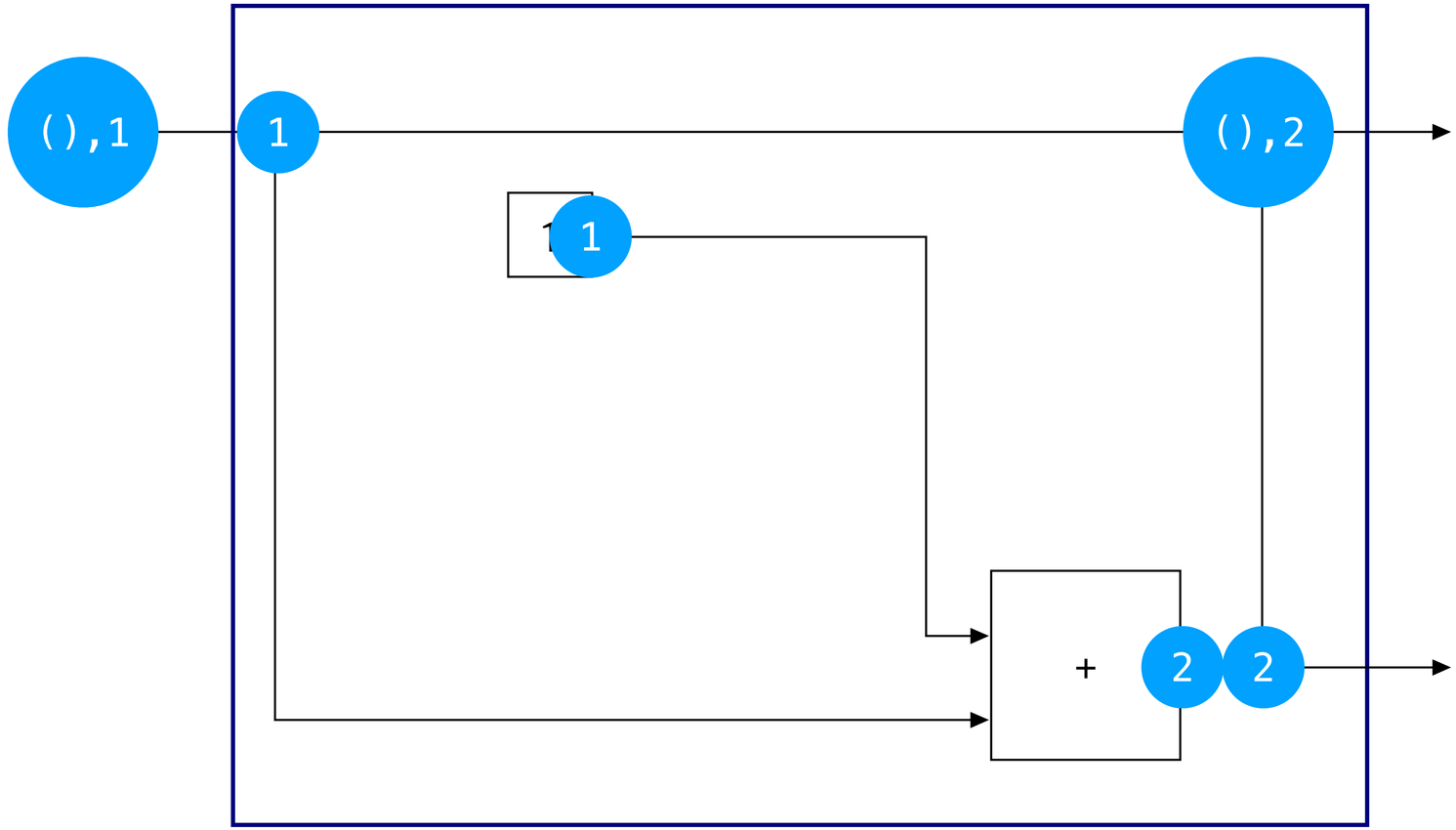
- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



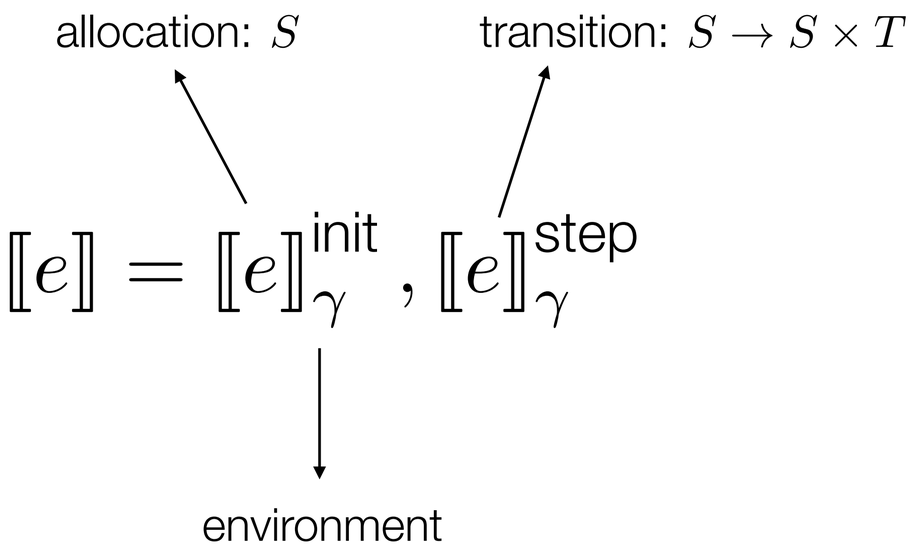
Example

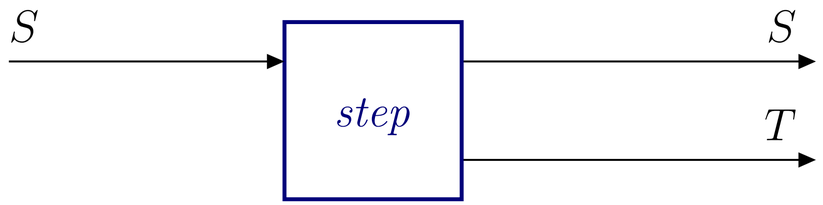
```
rec x = 1 + pre x
```

- Initial state: `(), 0`
- Output: `1, 2, 3, 4, ....`



Deterministic streams





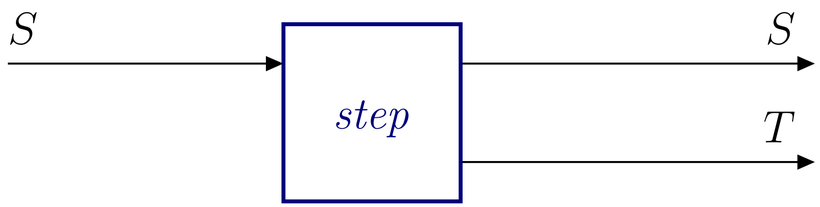
$$
\begin{array}{lcl}
[\![c]\!]_\gamma^{\text{init}} & = & ( ) \\
[\![c]\!]_\gamma^{\text{step}}\,(( )) & = & ( ), c \\
 & & \\
[\![x]\!]_\gamma^{\text{init}} & = & ( ) \\
[\![x]\!]_\gamma^{\text{step}}\,(( )) & = & ( ), \gamma(x) \\
 & & \\
[\![i \to \texttt{pre}\ e]\!]_\gamma^{\text{init}} & = & (i, [\![e]\!]_\gamma^{\text{init}}) \\
[\![i \to \texttt{pre}\ e]\!]_\gamma^{\text{step}}\,(p, s) & = & \textit{let}\ s', v = [\![e]\!]_\gamma^{\text{step}}\,(s)\ \textit{in}\ (v, s'), p
\end{array}
$$

Deterministic vs. probabilistic

Deterministic streams

Transition function returns a pair of state and value

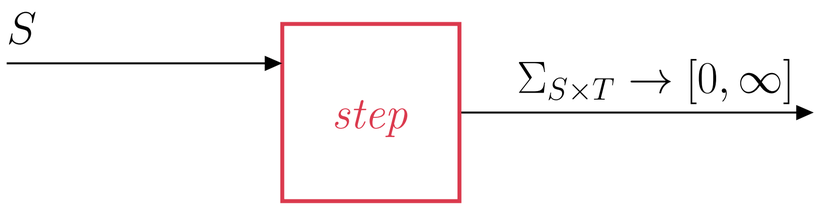
$$\mathit{CoStream}(T, S) = S \times (S \to S \times T)$$



Probabilistic streams

Transition function returns a **measure** over (state, value)

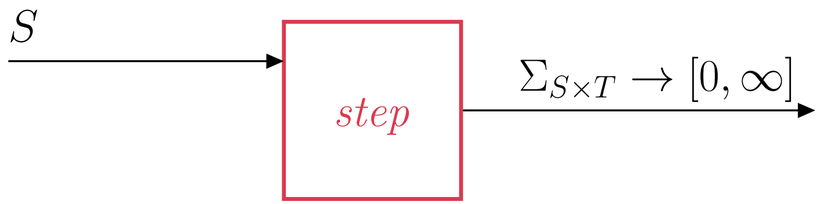
$$\mathit{CoPStream}(T, S) = S \times (S \to \Sigma_{S \times T} \to [0, \infty])$$



Probabilistic streams

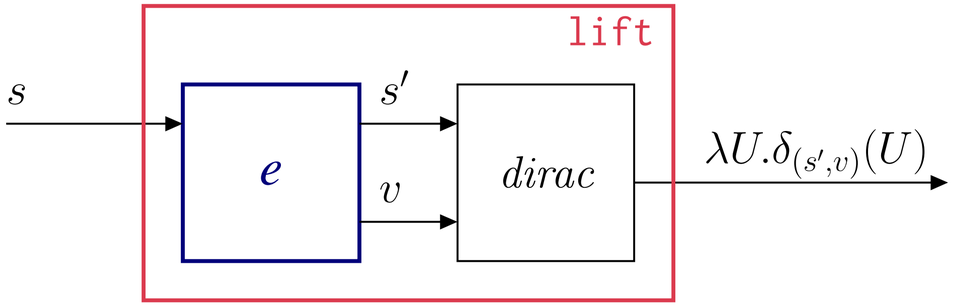
Transition function returns a *measure*

$CoPStream(T, S) = S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



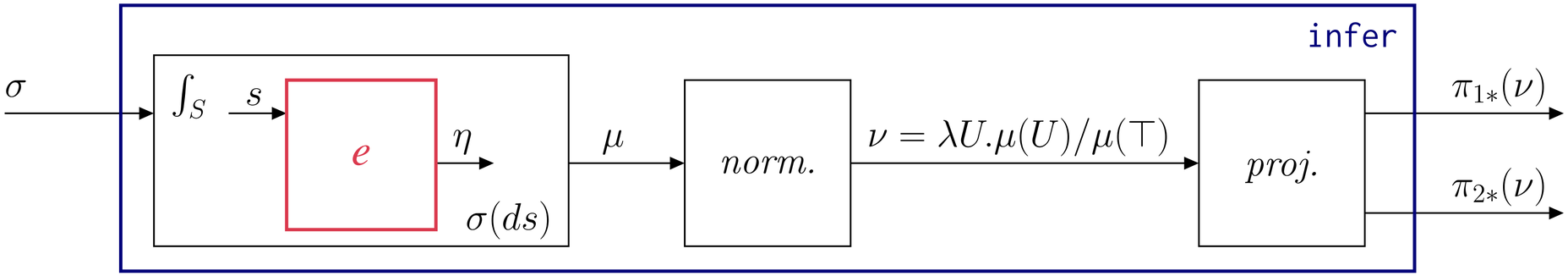
`lift` turns a deterministic expression into a probabilistic one

`lift:` $S \times (S \to S \times T) \to S \times (S \to \Sigma_{S \times T} \to [0,\infty])$



`infer` turns probabilistic expressions to a pair of distributions

`infer:` $S \times (S \to \Sigma_{S \times T} \to [0,\infty]) \to S$ `dist` $\times (S$ `dist` $\to S$ `dist` $\times T$ `dist`$)$



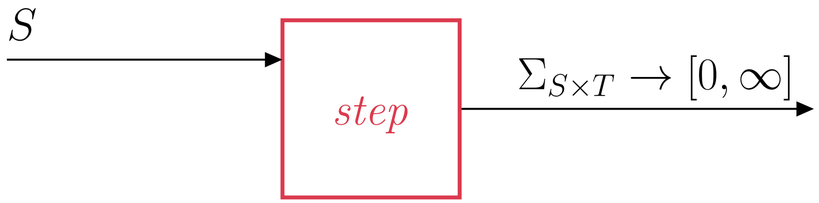
Probabilistic streams

allocation: S

transition: $S \to \Sigma_{S \times T} \to [0, \infty]$

$$\{\!|e|\!\} = \{\!|e|\!\}_\gamma^{\text{init}}, \{\!|e|\!\}_\gamma^{\text{step}}$$

environment



if e is deterministic

$$\{\!|e|\!\}_\gamma^{\text{init}} = [\![e]\!]_\gamma^{\text{init}}$$

$$\{\!|e|\!\}_\gamma^{\text{step}}(s) = \textit{let } s', v = [\![e]\!]_\gamma^{\text{step}}(s) \textit{ in } \delta_{s',v}$$

$$\{\!|\texttt{sample}(e)|\!\}_\gamma^{\text{init}} = [\![e]\!]_\gamma^{\text{init}}$$

$$\{\!|\texttt{sample}(e)|\!\}_\gamma^{\text{step}}(s) = \textit{let } s', \mu = [\![e]\!]_\gamma^{\text{step}}(s) \textit{ in } \int \mu(dv)\ \delta_{s',v}$$

$$\{\!|\texttt{observe}(e_1, e_2)|\!\}_\gamma^{\text{init}} = [\![e_1]\!]_\gamma^{\text{init}}, [\![e_2]\!]_\gamma^{\text{init}}$$

$$\{\!|\texttt{observe}(e_1, e_2)|\!\}_\gamma^{\text{step}}(s_1, s_2) = \textit{let } s_1', \mu = [\![e_1]\!]_\gamma^{\text{step}}(s_1) \textit{ in } \textit{let } s_2', v = [\![e_2]\!]_\gamma^{\text{step}}(s_2) \textit{ in } \mu_{\text{pdf}}(v) * \delta_{(s_1', s_2'),()}$$

Streaming inference

Reactive Probabilistic Programming

Particle filter

```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

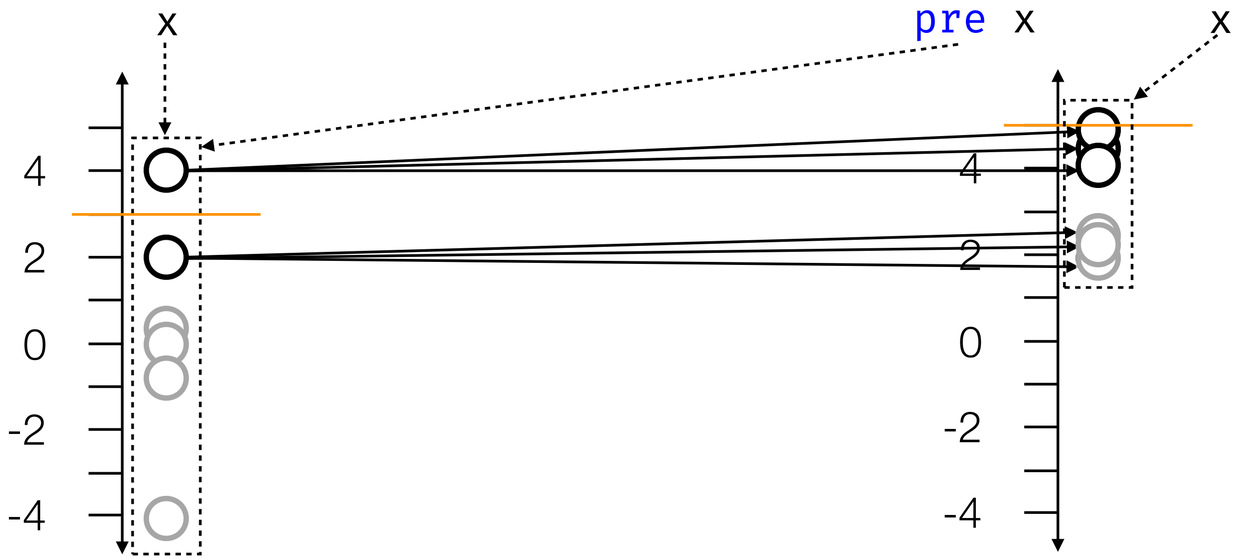
```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```

$t = 1$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), 5)
```

$t = 2$

```
sample (gaussian (pre x, 1))
observe (gaussian (x, 1), …)
```



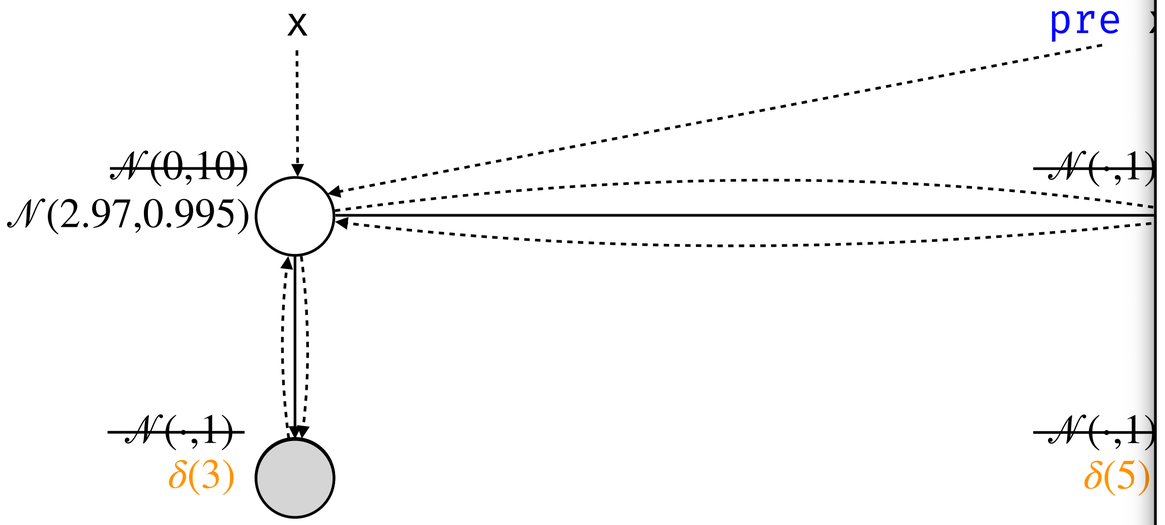
...

Delayed sampling

```
let proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10)) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

$t = 0$

```
sample (gaussian (0, 10))
observe (gaussian (x, 1), 3)
```



$t = 1$

```
sample (gauss
observe (gaus
```

pre x

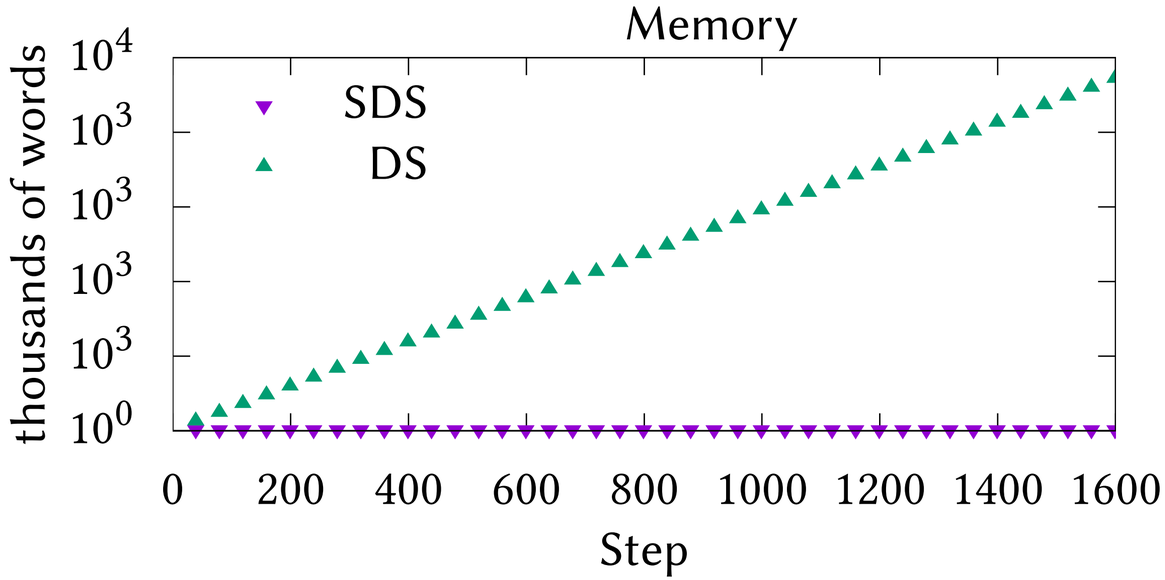
$\mathcal{N}(\cdot,1)$

$\mathcal{N}(\cdot,1)$

$\delta(5)$

$t = 2$

Unbounded resources



Benchmarks

Time to reach a target accuracy

Baseline:
SDS with 1,000 particles

SSI computations cost more,
but can outperform DS

	PF		DS		SSI	
MODEL	# PART.	TIME (MS)	# PART.	TIME (MS)	# PART.	TIME (MS)
Beta-Bernoulli	200	23.05 (22.54-23.90)	✓ 1	0.28 (0.27-0.28)	✓ 1	0.65 (0.65-0.66)
Gaussian-Gaussian	3000	877.44 (689.04-890.68)	150	62.99 (61.69-65.86)	150	182.57 (181.40-183.54)
Kalman-1D	15	3.27 (3.26-3.28)	✓ 1	0.33 (0.33-0.34)	✓ 1	1.15 (1.14-1.15)
Outlier	700	222.27 (220.76-223.84)	65	43.81 (43.45-46.48)	65	125.88 (125.27-128.08)
Robot	85	771.32 (767.98-775.51)	✓ 1	91.44 (90.94-92.07)	✓ 1	96.40 (96.21-97.53)
SLAM		✗	800	2812.55 (2755.99-2853.89)	800	5649.30 (5619.59-5675.81)
MTT		✗	60	2889.11 (2615.76-3244.30)	60	4457.79 (4068.35-4996.20)
Tree	150	35.55 (35.41-35.68)	90	58.83 (58.55-59.74)	✓ 1	2.67 (2.66-2.70)
Wheels	550	246.48 (245.06-248.75)	550	699.12 (672.25-713.64)	✓ 1	8.04 (8.00-8.10)
Delayed GPS	150	1221.00 (1218.76-1230.67)	9	304.73 (303.17-306.31)	✓ 1	108.55 (108.02-109.07)

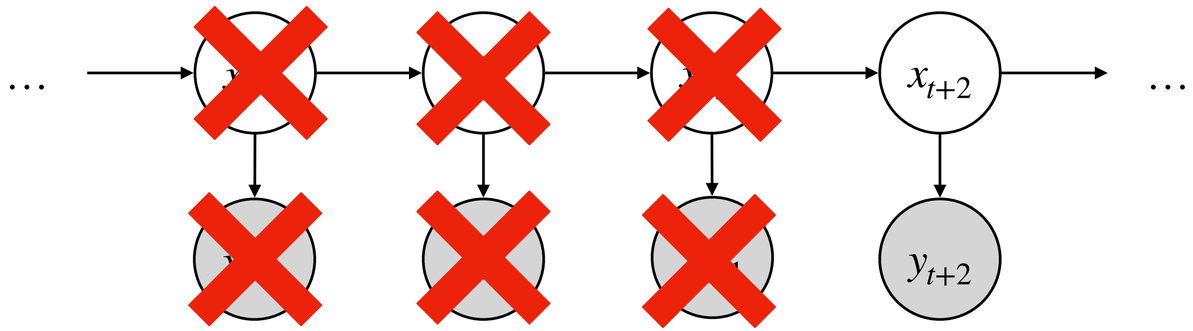
Trade-off: cost of symbolic computations vs. precision

Static Analysis

Reactive Probabilistic Programming

Bounded Memory Delayed Sampling?

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

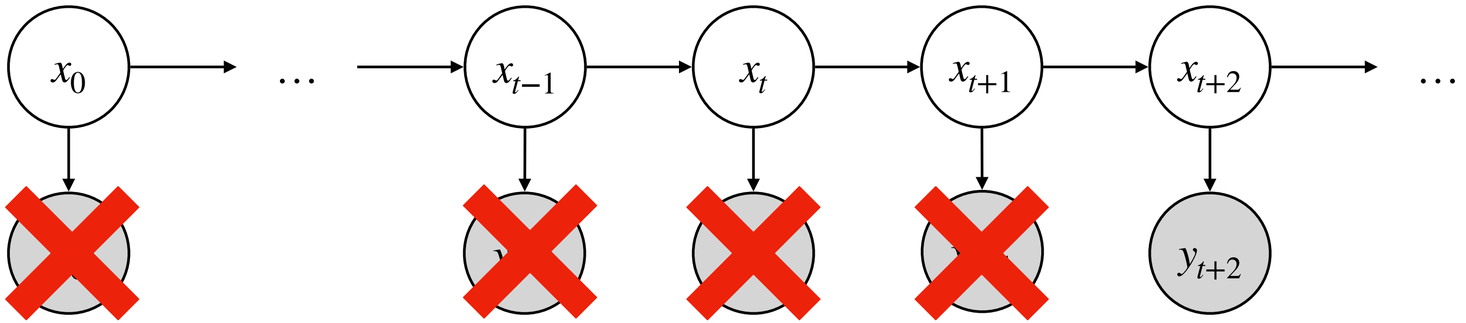


Yes!

Bounded Memory Delayed Sampling?

```
proba tracker (y) = x, x0 where
  rec init x0 = sample (gaussian (0, 10))
  and x  = x0 → sample (gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```

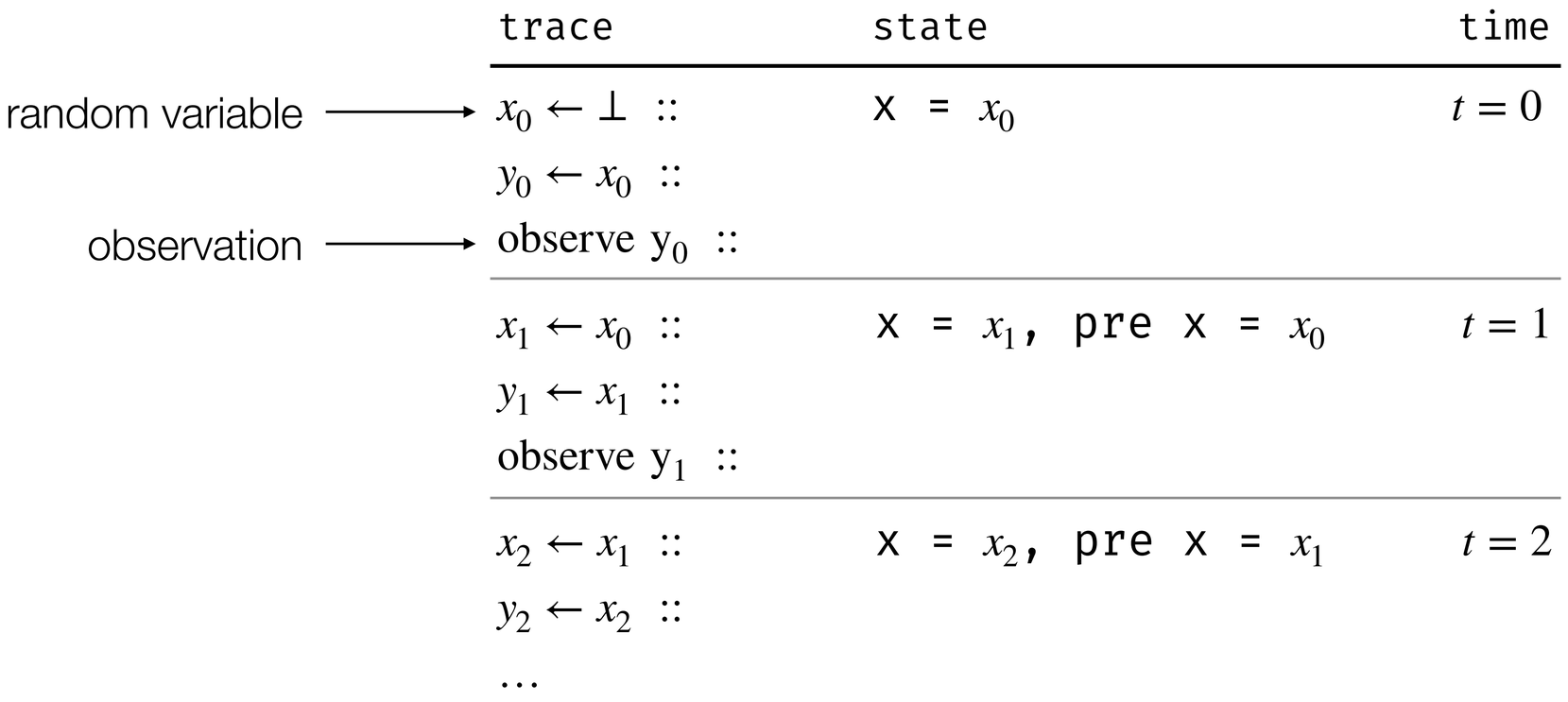
Can we determine if a given program will run in bounded memory?



No!

Trace: Abstract Execution

```
proba tracker (y) = x where
  rec x  = sample (gaussian (0, 10) → gaussian (pre x, 1))
  and () = observe (gaussian (x, 1), y)
```



Static Analysis for Delayed Sampling

Semantic properties

m-consumed property
Chains of variables before an observe are bounded

unseparated paths property
Chains of variables referenced in the state are bounded

Theorem: *The program satisfies these two properties iff it executes in bounded memory*

Static analysis

Track variables introduced but not used yet

Track maximal path between pairs of variable in the state

Theorem: *The program pass the analysis if it executes in bounded memory*

m-consumed Property

```
proba tracker (y) = x where
  rec x   = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables before an observe are bounded

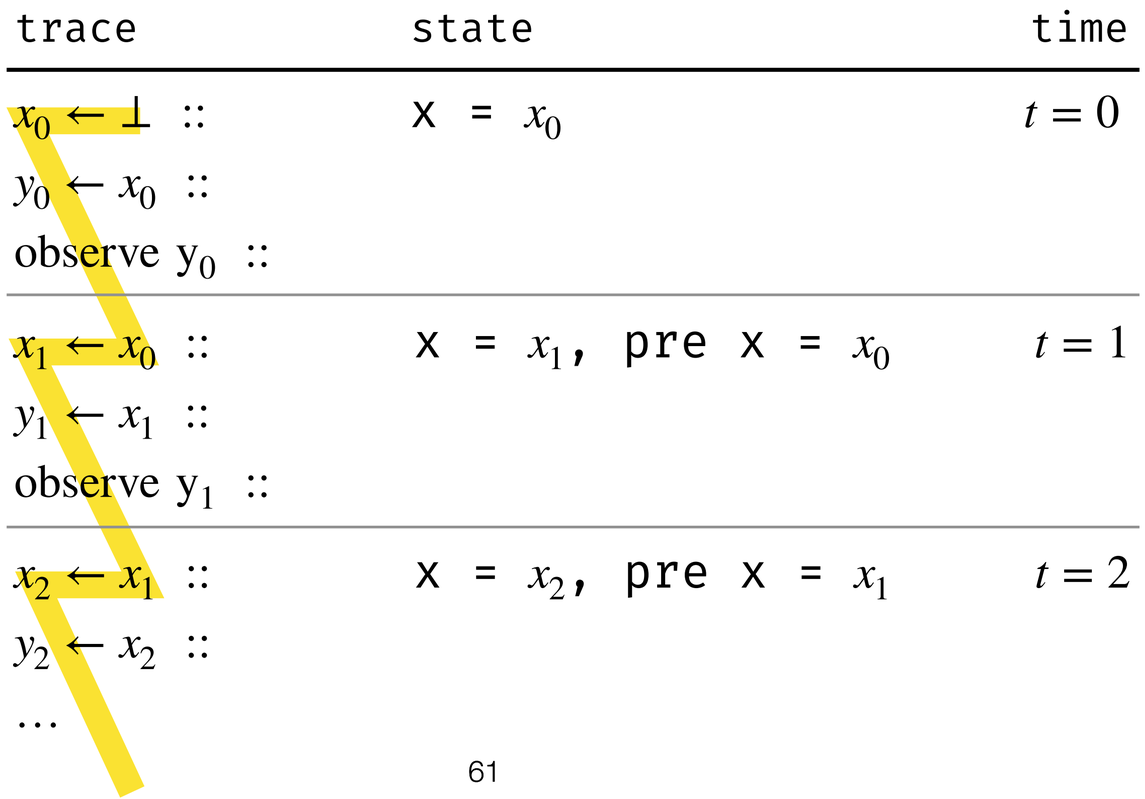
	trace	state	time
	$x_0 \leftarrow \bot$ ::	x = x_0	$t = 0$
x_0 is 1-consumed →	$y_0 \leftarrow x_0$ ::		
y_0 is 0-consumed →	observe y_0 ::		
	$x_1 \leftarrow x_0$ ::	x = x_1, pre x = x_0	$t = 1$
x_1 is 1-consumed →	$y_1 \leftarrow x_1$ ::		
y_1 is 0-consumed →	observe y_1 ::		
	$x_2 \leftarrow x_1$ ::	x = x_2, pre x = x_1	$t = 2$
	$y_2 \leftarrow x_2$ ::		
	…		

Yes!

Unseparated Paths Property

```
proba tracker (y) = x where
  rec x   = sample (gaussian (0, 10) → gauss
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded



Yes!

Unseparated Paths Property

```
proba tracker (obs) = x where
  rec init x0 = sample (gaussian (0, 10))
  and x  = x0 → sample (gaussian (pre x, 1)
  and () = observe (gaussian (x, 1), y)
```

Chains of variables referenced in the state are bounded

trace	state	time
$x_0 \leftarrow \bot$ ::	x = x_0	$t = 0$
$y_0 \leftarrow x_0$ ::	x0 = x_0	
observe y_0 ::		
$x_1 \leftarrow x_0$ ::	x = x_1, pre x = x_0	$t = 1$
$y_1 \leftarrow x_1$ ::	x0 = x_0	
observe y_1 ::		
$x_2 \leftarrow x_1$ ::	x = x_2, pre x = x_1	$t = 2$
$y_2 \leftarrow x_2$ ::	x0 = x_0	
…		

No!

References

Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs.
Lawrence Murray, Daniel Lundén, Jan Kudlicka, David Broman, Thomas B. Schön
AISTATS 2017

Semi-Symbolic Inference for Efficient Streaming Probabilistic Programming
Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, Michael Carbin
OOPSLA 2022

Reactive probabilistic programming
Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, Michael Carbin
PLDI 2020
https://github.com/IBM/probzelus

Statically Bounded-Memory Delayed Sampling for Probabilistic Streams
Eric Atkinson, Guillaume Baudart, Louis Mandel, Charles Yuan, Michael Carbin
OOPLSA 2021